

Efficient Resource Sharing Through GPU Virtualization on Accelerated High Performance Computing Systems[☆]

Teng Li^{a,*}, Vikram K. Narayana^a, Tarek El-Ghazawi^a

^a*NSF Center for High-Performance Reconfigurable Computing (CHREC), Department of Electrical and Computer Engineering, The George Washington University, 801 22nd Street NW, Washington, DC, 20052, USA*

Abstract

The High Performance Computing (HPC) field is witnessing a widespread adoption of Graphics Processing Units (GPUs) as co-processors for conventional homogeneous clusters. The adoption of prevalent Single-Program Multiple-Data (SPMD) programming paradigm for GPU-based parallel processing brings in the challenge of resource underutilization, with the asymmetrical processor/co-processor distribution. In other words, under SPMD, balanced CPU/GPU distribution is required to ensure full resource utilization. In this paper, we propose a GPU resource virtualization approach to allow underutilized microprocessors to efficiently share the GPUs. We propose an efficient GPU sharing scenario achieved through GPU virtualization and analyze the performance potentials through execution models. We further present the implementation details of the virtualization infrastructure, followed by the experimental analyses. The results demonstrate considerable performance gains with GPU virtualization. Furthermore, the proposed solution enables full utilization of asymmetrical resources, through efficient GPU sharing among microprocessors, while incurring low overhead due to the added virtualization layer.

Keywords: GPU, Virtualization, Resource Sharing, SPMD, Heterogeneous Computing, High Performance Computing

1. Introduction

Recent years have seen the proliferation of Graphics Processing Units (GPUs) as application accelerators in High Performance Computing (HPC) Systems, due to the rapid advancements in graphic processing technology over the past few years and the introduction of programmable processors in Graphics Processing Units (GPUs), which is also known as GPGPU, or General-Purpose Computation on Graphic Processing Units [2]. As a result, a wide range of HPC systems have incorporated GPUs to accelerate applications by utilizing the unprecedented floating point performance and massively parallel processor architectures of modern GPUs, which can achieve unparalleled floating point performance in terms of FLOPS (Floating-point Operations Per Second) up to the TeraFLOP barrier [3] [4]. Such systems range from clusters of compute nodes to parallel supercomputers. While examples of GPU-based computer clusters can be found in academia for research purpose, such as [5] and [6]. The contemporary offerings from supercomputer vendors have begun to incorporate professional GPU computing cards into the compute blades of their parallel computer products; example include the latest Cray XK7 [7] and SGI Altix UV [8] supercomputers. Yet another notable example is the Titan supercomputer [9] currently ranking the 2nd in the Top 500 supercomputer list [10]. Titan is equipped with 18,688 NVIDIA Tesla GPUs and is thereby able to achieve a sustained 17.59 PFLOPS LINPACK performance [10].

[☆]A preliminary version of this paper has been presented at 2011 IEEE International Conference on Parallel Processing (ICPP'11) [1].

*Corresponding author

Email addresses: tengli@gwu.edu (Teng Li), vikramkn@ieee.org (Vikram K. Narayana), tarek@gwu.edu (Tarek El-Ghazawi)

Table 1: GPU-based Supercomputers in The Top 30 List

Supercomputer (Ranking)	# of CPU Cores	# of GPUs	CPU/GPU Ratio
Titan (2 nd)	299,008	18688	16
Tianhe-1A (10 th)	102,400	7,168	14.3
Nabulae (16 th)	55,680	4,640	12
Tsubame2.0 (21 st)	17,984	4,258	4.2

Development of parallel applications for any of these GPU-based heterogeneous HPC systems requires the use of parallel programming techniques composed of both CPU and GPU parallel programs to fully utilize the computing resources. Among the different parallel programming approaches, the most commonly followed programming approach is the Single Program Multiple Data (SPMD) model [11]. Under the SPMD model, multiple processes execute the same program on different CPU cores, simultaneously operating on different data sets in parallel. Techniques of messaging passing such as MPI [12] are often deployed to achieve the SPMD parallelism effectively with necessary required inter-processor communication. By allowing autonomous execution of processes at independent points of the same program, SPMD serves as a convenient yet powerful approach for efficiently making use of the available hardware parallelism.

With the introduction of hardware accelerators, such as GPUs, as co-processors, HPC systems are exhibiting architectural heterogeneity that has given rise to programming challenges not previously existing in traditional homogeneous platforms. With the SPMD approach used for programming most of the homogeneous parallel architectures, directly offloading the program instances on to GPUs is not feasible due to the different Instruction Set Architectures (ISAs) of CPUs and GPUs. Moreover, GPUs are primarily suited for the Compute-Intensive portions of the program, serving as co-processors to the CPUs in order to accelerate these sections of the parallel program. The “single program” requirement of SPMD therefore means that every program instance running on the CPUs must necessarily have access to a GPU accelerator. In other words, it is necessary to maintain a one-to-one correspondence between CPUs and GPUs, that is, the number of CPU cores must equal the number of GPUs. However, due to the proliferation of many-core microprocessors in HPC systems, the number of CPU cores generally exceeds the number of GPUs, which is true for all GPU-based supercomputers in the top 20 list, as shown in Table 1 [10] [9] [13] [14] [15]. Therefore, the problem of system computing resources underutilization with the SPMD approach is general across GPU-based heterogeneous platforms and urgently needs to be solved with the increasing number of CPU cores in a single CPU due to the fast advancement of multi/many core technologies.

However, even though the number of physical GPUs can not match the number of CPU cores found in contemporary HPC systems, modern high-end GPU architecture is designed as massively parallel and composed of up to thousands of processing cores [16]. Moreover, since GPU programs are composed of parallel threads executed in parallel on these many processing cores physically, it is possible to achieve execution concurrency of multiple GPU programs on the single GPU. The Fermi [17] architecture from NVIDIA consisting up 512 Streaming Processor (SP) cores allows concurrent execution of up to 16 GPU kernels [18]. The increasing parallel computation capabilities of modern GPUs enables the possibility of sharing a single GPU to compute different applications or multiple instances of the same application, especially when the application problem size and parallelism is significantly smaller than the inherent parallelism capacity of the GPU.

In this paper, targeting at the problem of resource underutilization under SPMD model, we propose the concept of efficiently sharing the GPU resources among the microprocessor cores in heterogeneous HPC systems, by providing a virtualized unity ratio of GPUs and microprocessors. We develop the GPU resource virtualization infrastructure as the required solution for asymmetrical resource distribution. The GPU virtualization infrastructure provides the required symmetry between GPU and CPU resources by virtually increasing the number of GPU resources, thereby enabling efficient SPMD execution. In the meantime, we also theoretically analyze the GPU program execution within a single compute node, and provide analytical execution models for our resource virtualization scenarios, which provide different extents of GPU kernel concurrency support. Furthermore, we conduct a series of experiments using our virtualization infrastructure

on our NVIDIA GPU cluster node as verifications of the proposed modeling analysis as well as demonstration of well-improved process-level GPU sharing efficiency using the proposed virtualization approach.

The rest of this paper is organized as follows. Section 2 provides an overview of related work on GPU resource sharing in heterogeneous HPC systems and GPU virtualization under different contexts. A background of current GPU architectural and programming model is given in Section 3, followed by a formal analysis of the GPU execution models for our proposed virtualization solution in Section 4. Section 5 then discusses the implementation details of the GPU resource sharing and virtualization infrastructure. The experimental results are presented and discussed in Section 6, followed by our conclusions in Section 7.

2. Related Work

With the continued proliferation of GPGPU in the HPC field, virtualization of GPUs as computing devices is gaining considerable attention in the research community. One stream of recent research has focused on providing access to GPU accelerators within virtual machines (VMs). [19] [20] [21] Research studies in this stream focus on providing native CUDA programmability support within VMs, on computing systems equipped with NVIDIA GPUs. Gupta et al. [19] presented their GViM software architecture, which allows CUDA applications to execute within VMs under the Xen virtual machine monitor. The GPU is controlled by the management OS, called “dom0” in Xen parlance. The actual GPU device driver therefore runs in dom0, while applications execute on the guest OS or VM. By using a split-driver approach, the CUDA run-time API function calls from the application are captured by an interposer library within the VM, followed by transfer of the function parameters to dom0 for actual interaction with the GPU. A similar approach is adopted in vCUDA by Shi et al. [20], also on Xen. They additionally provide suspend and resume facility by maintaining a record of the GPU state within the virtualization infrastructure in the guest OS as well as the management OS domain.

However, the primary drawback of providing GPU access by using Xen is that NVIDIA drivers do not officially support Xen and CUDA drivers with version later than 2.2 do not work under Xen, which therefore makes the aforementioned approach unportable. Giunta et al. [21] circumvent this problem by using the Kernel Virtual Machine (KVM) available in Linux distribution, while following a similar split-driver approach coupled with API call interception. Their software infrastructure, termed gVirtuS, focuses on providing GPU access to VMs within virtual clusters, as part of a cloud computing environment. For cases when the host machine does not have a local GPU, they envision that the virtual machines will use TCP/IP to communicate with other hosts within the virtual cluster in order to gain access to remote GPUs.

While achieving GPU virtualization by providing CUDA support within VMs appears to be an attractive solution for our problem, launching multiple VMs within the same compute node can result in significant overheads for HPC applications targeting at performances. To elaborate, efficient SPMD execution requires all CPU cores (processes) to have a virtual view of a GPU; with the VM approach, this would require a VM to be launched for every CPU core within the compute node. Since the number of CPU cores per node is rapidly on the rise, significantly increasing overheads with the VM-based approach for GPU sharing are inevitable. Further, all the available VM-based GPU sharing approaches time-share the GPU. Thus, the potential for simultaneous executing GPU functions from multiple processes can not be exploited.

Other types of solutions have also been proposed for GPU virtualization in HPC systems. For example, Duato et al [22] propose the use of remote GPU access similar to [21], for cases when high performance clusters do not have a GPU within every compute node. Instead of using a VM, they propose a GPU middleware solution consisting of a daemon running on GPU-capable nodes that serves requests from non-GPU node clients. The client nodes incorporate a CUDA wrapper library to capture and transfer API function calls to the server using TCP/IP. Although the VM overheads are removed, their proposed solution can result in communication overheads in accessing GPUs from remote compute nodes. Moreover, simultaneous execution of multiple GPU kernels is not discussed.

Another stream of research has concentrated on GPU sharing to eliminate resource under-utilization. Guevara et al. [23] propose an approach that involves run-time interception of GPU invocations from one or more processes, followed by merging them into one GPU execution context. Currently their solution

is demonstrated for two kernels, with the merged kernels predefined manually. Similarly, Saba et al. [24] presented an algorithm that allocates GPU resources for tasks based on the resource goals and workload size. While they target at a different problem of a time bound algorithm that optimizes the execution path and output quality, the employed GPU sharing approach is to merge the kernels similarly. Although kernel merging can be useful for SPMD execution, it would need compiler support for generation of the combined kernels a priori, which can be avoided by using the concurrent kernel execution support from Fermi. Moreover, with merged kernels, multiple kernels are invoked simultaneously, therefore it does not exploit the possibility to hide data transfer overhead with kernel execution, which can be achieved through concurrent kernel execution and data transfer from current GPU architecture. Similar in scheduling purpose as [23], our previous work [25, 26, 27] described an off-line scheduling framework to achieve improved performance and resource utilization of many GPU tasks with possible inter-task concurrencies through GPU sharing. While [25] targets at a completely different problem, it utilizes the similar analytical GPU sharing approach as our proposed work in this paper. Meanwhile, GPU resource under-utilization has also been studied in cloud computing. Ravi et al. [28] proposed a GPU sharing framework for the cloud environment, base on which they provided a GPU kernel consolidation algorithm to combine kernel workloads within the cloud and thus achieves improved throughput by utilizing both space and time sharing.

For thread-level GPU sharing, Peters et al. [29] proposed another technique for sharing an NVIDIA GPU among multiple host threads on a compute node. Their technique involves the use of persistent kernels that are initialized together within a single execution context, and remain executing on the GPU indefinitely. Each of these persistent kernels occupies a single thread block on the GPU, and they execute the appropriate function based on commands written to the GPU memory by the host process running on the CPU. This “management thread” accepts requests from other CPU threads for the use of GPU resources. Their approach allows for multiple kernels to simultaneously execute even on devices that do not support concurrent kernel execution. However, due to the persistent nature of the kernels, the number of thread blocks is severely limited, which means that the memory latency may not be hidden effectively. As a result, highly data-parallel applications will not be able to take full benefit of the GPU resources. Furthermore, their approach requires significant changes to the application code to fit within a thread block. Also, the communication mechanism between the management thread and other threads is not clear. Nevertheless, the use of a management process to control the GPU and manage the resources is a common feature shared with our proposed solution.

To provide process-level GPU sharing, our initial work [1] presented the GPU virtualization approach and prototype as the basis for our proposed work in this paper. Another solution is proposed by the S_GPU project [30], which provides a software layer that resides between the application and the GPU, typically used for GPU time-sharing between MPI processes in a parallel program. Each MPI process is provided the view of a private GPU, through a custom stream-based API. Each process inserts the GPU commands, such as memory copy, kernel launch, etc, in the required sequence into a stream object, irrespective of the number of GPUs available. When the process initiates the execution of a stream, all the enqueued GPU commands are then executed in the required sequence. The S_GPU software stack takes care of sharing the available GPUs among the streams from multiple processes. The approach followed by S_GPU is complementary to [1] and our approach here, and may be combined with our proposed approach by simultaneously executing kernels from multiple processes for efficient GPU sharing. Furthermore, the efficient GPU sharing approach proposed by [31, 32, 33] is relevant to this work as well.

On the other hand, virtualization of co-processors has been studied for other technologies as well, such as Field-Programmable Gate Arrays (FPGAs). For example, Huang and Hsiung [34] provide an OS-based infrastructure that allows multiple hardware functions from different processes to be configured in the FPGA, by using the partial run-time reconfiguration feature. Their virtual hardware mechanism allows a given hardware function to be used by multiple applications, as well as enabling one application to access multiple hardware functions simultaneously in the FPGA. Similar FPGA work also includes the RAD architecture proposed by [35]. Our previous work [36] also uses the partial run-time reconfiguration feature of the FPGA, albeit for enabling efficient SPMD execution in HPC systems. This work partitions the FPGA into multiple regions, and allocates each region to a CPU core within the compute node. By thus providing a virtual FPGA to every CPU core, the required 1:1 ratio of the number of CPU cores to the number of virtual FPGAs

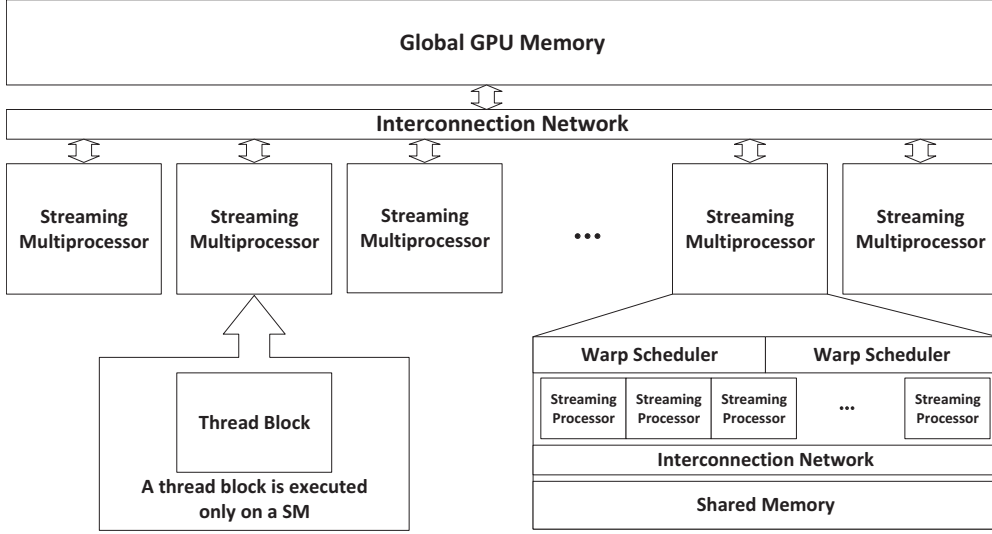


Figure 1: An Overview of Current GPU Architecture

is achieved. This work forms the basis of our proposed virtualization solution for GPU-based HPC systems. Our proposed approach effectively utilizes modern GPU features, such as concurrent kernel execution and concurrent data transfer and execution, to improve the execution performance of the system. Overheads are kept small by maintaining a simple communication mechanism between the CPU processes and our proposed virtualization infrastructure.

3. Background on GPU Programming and Architectural Model

A brief overview of GPU programming and computing architectural model is presented in this section. Our further modeling analysis and implementation are based on Compute Unified Device Architecture (CUDA) [18] and Open Computing Language (OpenCL) [37] on the programming model side as well as the NVIDIA GPU architecture [17] on the hardware architecture side.

3.1. Programming Models of Modern GPUs

As previously mentioned, CUDA and OpenCL are two prevalent GPU computing programming models provided by NVIDIA and Khronos Working Group, respectively. Both models follow Single Instruction Multiple Thread (SIMT) model by executing data-parallel GPU kernel functions within the GPU, while providing abstractions of thread group hierarchy and shared memory hierarchy. In terms of thread group hierarchy, both models (CUDA/OpenCL) provide three hierarchy levels: Grid/NDRange, Block/Workgroup and Thread/Workitem. For convenience, we will use CUDA terminology in the rest of this paper, namely, grids, blocks and threads. GPU kernels are launched per grid and a grid is composed of a number of blocks, which have access to the global device memory. Each block consists of a group of threads, which are executed concurrently and share the accesses to the on-chip shared memory. Each thread is a very lightweight execution of the kernel function. From programming perspective, the programmer needs to write the kernel program for one thread and decides the total number of threads to be executed on the GPU device while dividing the threads into blocks based on the data-sharing pattern, memory sharing and architectural considerations.

3.2. Architectural Model of Modern GPUs

Modern GPUs are composed of massively parallel processing units and hierarchies of memories. From the architectural perspective of modern GPU hardware, the GPU architecture (Fermi [17]) from NVIDIA, as



Figure 2: The GPU Execution Cycle for a Single Process

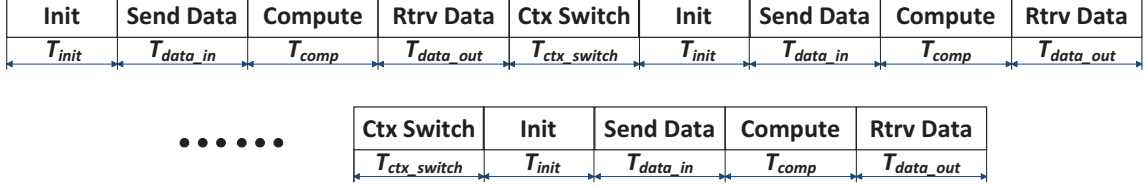


Figure 3: The Native GPU Execution Model from Multiple Processes without Virtualization

shown in Figure 1, is composed of up to 16 Streaming Multiprocessors (SMs), each of which can be further decomposed of 32 Streaming Processor (SP) cores as the second processing element hierarchy. Memory hierarchies are composed of the global device memory shared by all SMs and private shared memories of each SM that are shared by all SPs within each SM. Each thread is executed on an SP core and each block runs on an SM at a given time. While each SP holds a number of registers for each thread and executes threads sequentially; within each SM, threads are scheduled as a batch of 32 threads called a warp. Two warp schedulers exist in an SM and multiple warps are allowed to co-exist on the same SM. Especially when threads have high memory access latency, increasing the warp occupancy by having multiple warps co-exist on a single SM simultaneously can improve the overall execution performance.

3.3. Kernel-level Execution Flow of Modern GPUs

Upon the launching of kernel composed of multiple blocks, each block can only be executed on an SM. Multiple blocks can only reside on the same SM provided there are enough SM resource, which includes the register, shared memory and total warps can be scheduled in an SM. Since each SM has limited number of registers, a fixed size of shared memory and a maximum number of warps that can co-exist and be scheduled, only multiple blocks meeting these constraints can be scheduled within a single SM. However, blocks are only limited to a single kernel. In other words, blocks from different kernels can not be concurrently scheduled in conventional GPU devices. Nevertheless, current CUDA devices with computing capability higher than 2.0 (Fermi or later) support concurrent kernel execution, which allows different kernel to be launched from the same process (GPU program) using CUDA streams. Concurrent kernel execution allows blocks from different kernels to be scheduled simultaneously. Furthermore, by using asynchronous CUDA streams, concurrent data transfer and kernel execution can also be achieved among multiple streams, each of which carries a GPU kernel. Note, however, that GPU kernels launched from independent CPU processes cannot be concurrently executed, which necessitates the proposed virtualization approach and implementation described in the following sections.

4. Process-level Sharing Modeling and Performance Analysis with GPU Virtualization

In this section, we perform a theoretical study on the performance of process-level GPU sharing. We describe and analyze the achievable GPU sharing efficiency when our virtualization approach is applied. Our analysis is based on the actual GPU sharing scheme provided by GPU virtualization. Here we first perform a study of the factors that affect the overall performance of multiple GPU kernel executions. By conducting this study, we will be able to investigate the performance potential of using our proposed concept of centralized daemon-process-based GPU sharing. We will first layout the conventional scheme when multiple processes share a single GPU, followed by potential improvement technique using GPU virtualization. Our analysis is primarily based on a series of GPU kernel execution models depicting different

Table 2: Parameters Defined for Analytical Modeling

Symbol	Definition
$N_{processor}$	The total number of processors in each node
$N_{process}$	The number of parallel SPMD processes (GPU tasks) running in each node, which should not exceed $N_{processor}$
N_{VGPU}	The number of virtual GPU resources for SPMD processes exposed from the virtualization layer, which equals to $N_{processor}$
T_{init}	The time for each process to initialize the GPU and corresponding resources
T_{ctx_switch}	The average GPU context-switch overhead associated with each process
T_{data_in}	The time for each process to transfer the input data into the GPU device memory
T_{data_out}	The time for each process to retrieve the result data back from the device memory
T_{comp}	The time for the GPU to compute the task
$T_{total_ci_ps1}$	The time to execute C-I kernels from all processes under PS-1 with virtualization
$T_{total_ci_ps2}$	The time to execute C-I kernels from all processes under PS-2 with virtualization
$T_{total_ioi_ps1}$	The time to execute IO-I kernels from all processes under PS-1 with virtualization
$T_{total_ioi_ps2}$	The time to execute IO-I kernels from all processes under PS-2 with virtualization
$T_{total_no_vt}$	The time to execute kernels from all processes without virtualization
S_{ci}	The theoretical speedup that can be achieved for C-I kernels
S_{ioi}	The theoretical speedup that can be achieved for IO-I kernels
S_{max_ci}	The theoretical maximum speedup that can be achieved for C-I kernels
S_{max_ioi}	The theoretical maximum speedup that can be achieved for IO-I kernels

process-level kernel execution scenarios. The proposed execution model considers multiple factors affecting the overall sharing efficiency and captures the timing of multiple kernel executions from many GPU requests on the process level (from many processors). Furthermore, we define necessary model parameters and derive the corresponding performance formula with performance comparison and analysis.

To study the potential GPU performance gain with the virtualization approach, we primarily focus our discussions on GPU tasks while excluding the CPU tasks. This is because we are mostly interested in enhancing the execution performance of many GPU tasks launched through SPMD programs. Meanwhile, we also try to avoid adding unnecessary complications to model the scenario of co-scheduling both CPU and GPU tasks.

4.1. Conventional Process-level GPU Request and Execution Model

Conventional, when multiple GPU kernel execution requests are launched from many SPMD processes, processes have direct access to the physical GPU without the proposed add-on virtualization layer. Here we base our analysis on a simplified scenario that a single GPU is requested from multiple processes running on multi-processors (cores) within each cluster node, due to the aforementioned asymmetrical CPU/GPU resource distribution. Under this scenario, multiple GPU requests from processors within each node will place individual GPU tasks/kernels on the shared single physical GPU simultaneously. However, current GPU device is only able to accept process-level requests one after another and thus sequentially execute multiple GPU kernels.

To analyze the execution pattern of conventional GPU sharing, we model the execution cycle for a given process to perform a GPU computing task in Figure 2. The modeled execution cycle is composed of the following four timing stages: the process first initializes the GPU device, creates its own GPU context and allocates the GPU device memory; followed by sending the input data, computing the task and retrieving the data back to the request process. Figure 2 also specifies the timing parameters for each stage of the execution cycle. Table 2 further defines necessary parameters used for the following modeling analysis.

Under conventional execution scheme within each node, processes treat the GPU as a non-virtualized resource. Current NVIDIA CUDA architecture allows multiple host processes to be executed native sharing

Listing 1: CUDA Stream Programming Style-1 (PS-1)

```
for (int i = 0; i < NumStream; i++)
    cudaMemcpyAsync (dst, src, count, cudaMemcpyHostToDevice, stream[i]);
for (int i = 0; i < NumStream; i++)
    myKernel <<<GridSize, BlockSize, ShareMemSize, stream[i]>>> (parameter);
for (int i = 0; i < NumStream; i++)
    cudaMemcpyAsync (dst, src, count, cudaMemcpyDeviceToHost, stream[i]);
```

Listing 2: CUDA Stream Programming Style-2 (PS-2)

```
for (int i = 0; i < NumStream; i++)
{
    cudaMemcpyAsync (dst, src, count, cudaMemcpyHostToDevice, stream[i]);
    myKernel <<<GridSize, BlockSize, ShareMemSize, stream[i]>>> (parameter);
    cudaMemcpyAsync (dst, src, count, cudaMemcpyDeviceToHost, stream[i]);
}
```

mode, by creating multiple GPU contexts for each process. The GPU kernels from multiple processes are serially executed in the order queued with respective GPU context-switch overheads. In other words, the execution model for conventional GPU sharing (with CUDA) among multiple processes is sequential among tasks with no concurrency possible. Moreover, each process needs an additional amount of time overhead to initialize the GPU device and its own resource such as context. Thus, in the execution model, we assume there is an average fixed GPU initialization overhead for each process, T_{init} , at the beginning. We also make assumption that there is an average context-switch overhead T_{ctx_switch} between every two processes, as shown in Figure 3. Therefore, the modeled total execution time for the conventional scheme without virtualization can be derived with Equation (1), which will be our performance comparison/analysis basis.

$$T_{total_no_vt} = N_{process}(T_{init} + T_{data_in} + T_{comp} + T_{data_out}) + (N_{process} - 1)T_{ctx_switch} \quad (1)$$

4.2. The Optimized Sharing Scheme and Execution Model Under GPU Virtualization

4.2.1. Streaming Kernel Execution with CUDA and Potential Concurrencies

As discussed earlier, GPU kernel execution from multi-processes achieves zero concurrency, however, current GPU technology (CUDA) is able to provide concurrencies of both kernel execution and I/O transfer through CUDA streams. However, the prevalent GPU Fermi architecture requires all CUDA streams to be launched simultaneously within a single GPU context to achieve both kernel and I/O concurrencies. In other words, if multiple kernels from SPMD processes can be launched through multiple CUDA streams from a single GPU context, the GPU can be efficiently shared among multiple kernels through streaming execution.

With streaming execution, three types of concurrencies are possible: (a) the execution of multiple concurrent kernels, (b) the concurrency between kernel execution and either GPU input or output data transfer, and (c) the concurrency between input data transfer and output data transfer. Currently, CUDA streams can be programmed with two styles [18] achieving different concurrency/overlapping behaviors (kernel concurrency vs. I/O concurrency). For simplicity, we here define that Programming Style-1 (PS-1) is primarily to achieve kernel execution concurrency while Programming Style-2 (PS-2) is to focus on I/O concurrency. Both definitions will be used in the following discussions. To demonstrate the difference, here we list two simple code snippets for both PS-1 and PS-2 in Listing 1 and 2 respectively. Under both programming styles, *NumStream* streams are launched. While the three stages: “Send Data”, “Compute” and “Rtrv Data” are executed as a batch for each single stream consecutively under PS-2, the same stages of all streams are executed as a batch under PS-1.

Under Fermi architecture, up to 16-way concurrency of kernel launches are permitted from individual streams. Upon being executed on the GPU, all CUDA streams are multiplexed into a single hardware work queue, as demonstrated in both Figure 4 and 5. Thus the two programming styles result in two different execution sequences inside the hardware work queue and thus different concurrencies due to data dependencies. We note that the three stages within each stream are all asynchronous operations. In

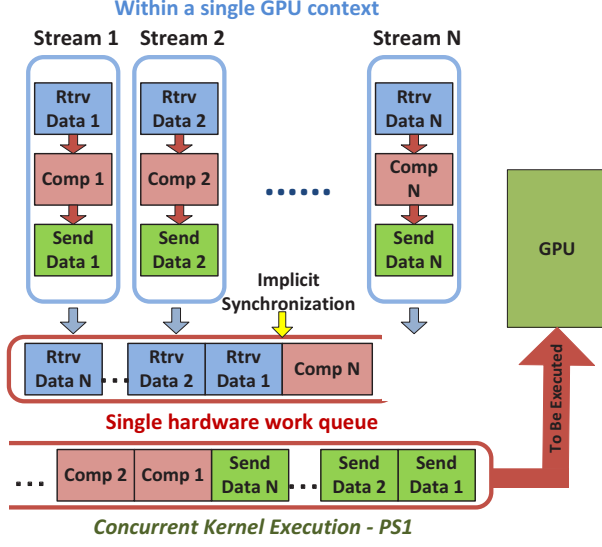


Figure 4: The Streaming Model for Kernel Concurrency

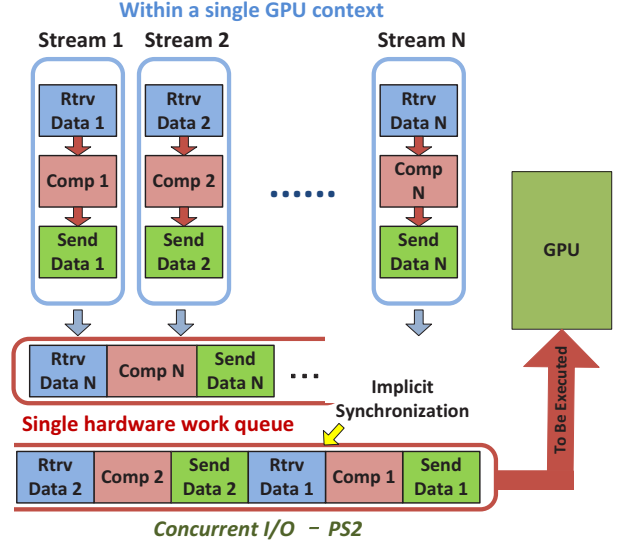


Figure 5: The Streaming Model for I/O Concurrency

other words, each operation is non-blocking provided there is no required data dependency and implicit synchronization.

Due to the separated for loops under PS-1, shown in List 1, “Send Data” from all streams are first queued into the hardware work queue, followed by each “Comp” from all streams, with each “Rtrv Data” followed at last, as shown in Figure 4. For PS-1, the big for loop shown in List 2 allows each stream sequentially following each other within the hardware work queue to be processed by the GPU.

Since we here mainly focus on an SPMD program, there is no data dependency among each of “Send Data i ” (i ranges from 1 to N), while this also applies among each of “Comp i ” as well as each of “Rtrv Data i ”. However, for each stage of a single stream i , “Comp i ” has the data dependency on the input data of “Send Data i ” while “Rtrv Data i ” depends on the result data from “Comp i ”. Therefore, PS-1 achieves different concurrency behavior from PS-2.

CUDA currently only provides limited stream support [18]. Specifically, operation requires an implicit synchronization (dependency check) to see if a kernel has finished (“Rtrv Data i ” for example) can: (1) start only when all prior kernel launches have started executing. (2) block all later kernel launches until checking on current kernel launch is completed. Thus, in Figure 4 under PS-1, “Rtrv Data 1” can only start after “Comp N ” and achieve very little concurrency with “Comp N ” while concurrencies can be achieved between “Send Data i ” and “Comp i ” as well as among all “Comp i ”s. Similarly, in Figure 5 under PS-2, “Rtrv Data i ” blocks “Comp $i+1$ ” due to the required dependency check. Therefore “Comp $i+1$ ” can only start after “Comp i ” is finished while “Send Data $i+1$ ” can still overlap with “Rtrv Data i ” and even “Comp i ”. Those overlapping behaviors serve as the basis for our execution model.

4.2.2. The Optimized Process-level GPU Sharing Scheme with GPU Virtualization

Considering the possible performance advantage with execution concurrency, here we propose the process-level GPU sharing scheme utilizing streaming execution. Under the optimized GPU sharing scheme, requests from multiple SPMD processes are launched using separate CUDA streams within a single GPU context. Our virtualization technique creates a single daemon process that handles GPU requests from multiple processes, as shown in Figure 6. Since the daemon process creates a single required GPU context, launching GPU kernel requests inside the daemon process with separate CUDA streams allows the potential concurrencies among multiple processes. Additionally, the optimized sharing scheme is able to avoid context-switch overheads

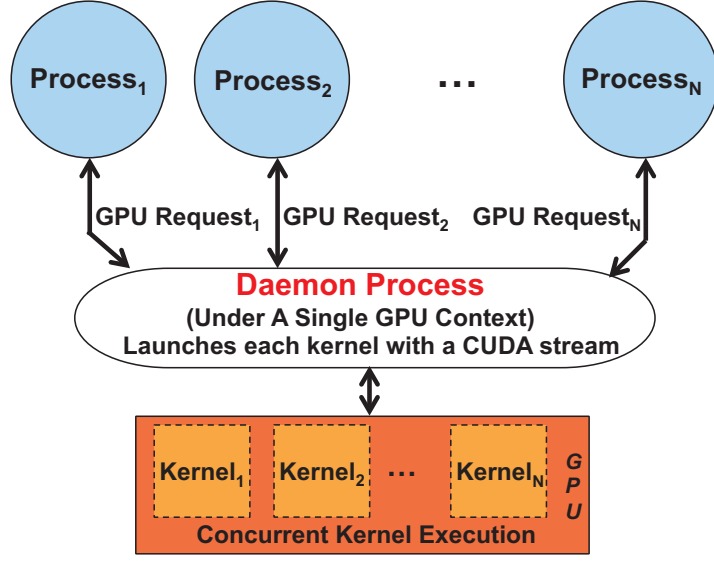


Figure 6: The Optimized Process-level GPU Sharing Scheme

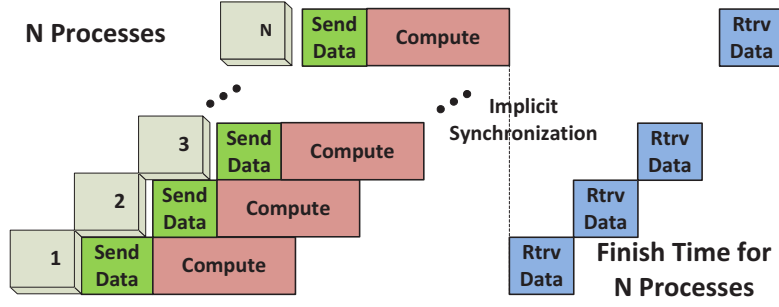


Figure 7: Compute-Intensive Kernels Under PS-1

among processes as well as hide the GPU initialization overhead, which is due to the single daemon process been created. Further virtualization implementation details will be discussed in the following section.

4.2.3. The GPU Execution Model Under the Optimized GPU Sharing Scheme

To study the potential performance gain with the proposed virtualization approach, we develop a series of execution models to demonstrate the overlapping behaviors and estimate the total GPU execution time. Since the execution model is to provide an estimated performance upper bound, we assume that the GPU resource is large enough to accommodate $N_{process}$ GPU kernels and also that single directional data transfers always take the full I/O bandwidth and therefore cannot be inter-overlapped. We also assume that process sequence is maintained in order since the process finishing order does not affect the total execution time. Since the daemon process is always running and has been initialized, T_{init} is a one-time overhead that can be hidden. Thus T_{init} is not considered in our analysis.

Considering the overlapping nature of two different stream programming styles, we consider two representative kernel cases: *Compute-Intensive* and *I/O-Intensive*. For simplicity, the kernel is considered *Compute-Intensive* when $T_{data_in} \leq T_{comp}$ and $T_{data_out} \leq T_{comp}$; and defined as *I/O-Intensive* when both T_{data_in} and T_{data_out} are greater than T_{comp} .

Figure 7 and 8 show the execution and overlapping behaviors of Compute-Intensive (C-I) case written with PS-1 and PS-2 respectively. As we analysed earlier, under PS-1 shown in Figure 7, C-I kernels can

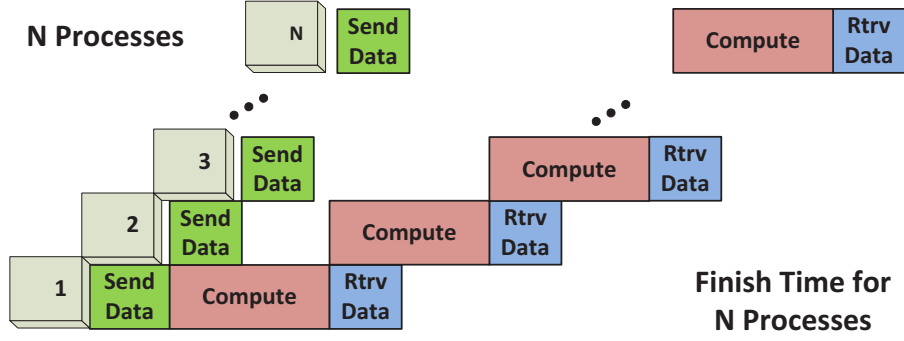


Figure 8: Compute-Intensive Kernels Under PS-2

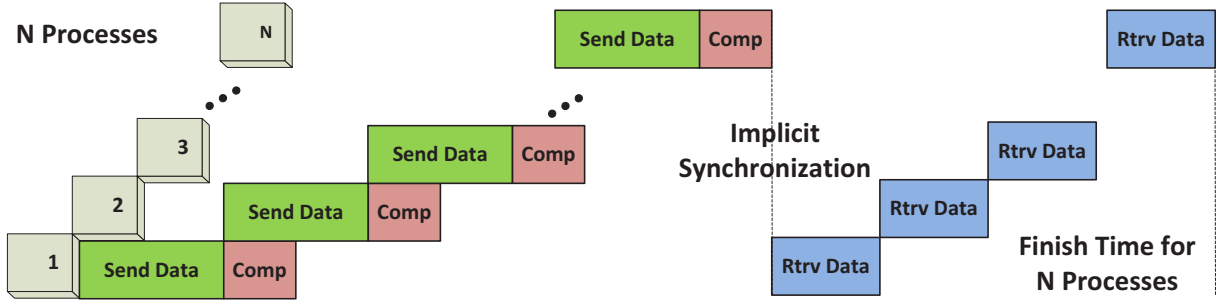


Figure 9: I/O-Intensive Kernels Under PS-1

overlap T_{comp} with both T_{data_in} and T_{comp} . Therefore, the total execution time can be represented in Equation (2).

$$T_{total_ci_ps1} = N_{process}(T_{data_in} + T_{data_out}) + T_{comp} \quad (2)$$

For C-I kernels programmed with PS-2, as shown in Figure 8, only I/O transfer can be inter-overlapped and overlapped with T_{comp} . T_{comp} cannot be inter-overlapped due to the implicit synchronization of T_{data_out} blocking the following T_{comp} . Thus we are able to derive the total execution time in Equation (3).

$$T_{total_ci_ps2} = T_{data_in} + N_{process}T_{comp} + T_{data_out} \quad (3)$$

By comparing Equation (2) and (3), we are able to see that $T_{total_ci_ps1} < T_{total_ci_ps2}$. Therefore, choosing PS-1 for C-I kernels is able to achieve better performance by concurrent kernel execution. We also note that the assumption of complete overlapping among all T_{comp} is merely to show the theoretical performance upper bound with the execution model.

Figure 9 and 10 demonstrate the execution scenario when I/O-Intensive (IO-I) kernels are programmed with PS-1 and PS-2 respectively. For IO-I kernels, the input data transfer of a subsequent kernel can only finish after the computation phase of the earlier kernel. In other words, the computation phases cannot overlap. When programmed with PS-1, the only possible overlapping is between T_{comp} and T_{data_in} . However, since T_{comp} is comparatively much smaller than T_{data_in} , the overlapping cannot achieve much performance improvement. The derived total execution time can be derived in Equation (4), which is the same as Equation (2)

$$T_{total_ioi_ps1} = N_{process}(T_{data_in} + T_{data_out}) + T_{comp} \quad (4)$$

Considering IO-I kernels programmed with PS-2, T_{data_in} can be overlapped with both T_{data_out} and T_{comp} , as shown in Figure 10. We note that Figure 10 only demonstrates the case when $T_{data_out} < T_{data_in}$. By

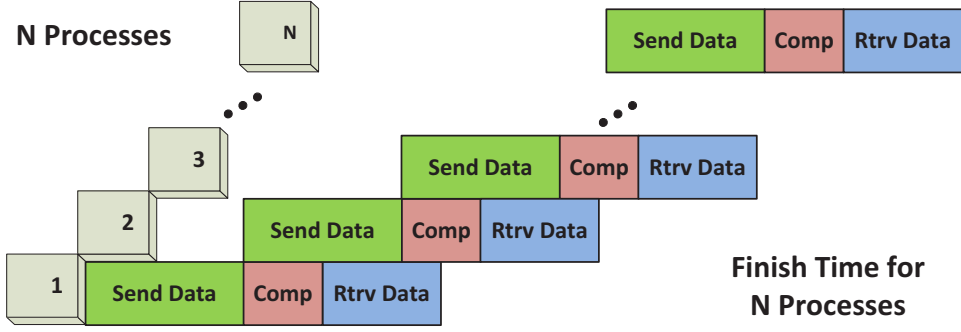


Figure 10: I/O-Intensive Kernels Under PS-2

also considering the other case when $T_{data_out} \geq T_{data_in}$, the total execution time can be derived accordingly with both Equation (5) and (6).

$$T_{total_ioi_ps2} = N_{process} T_{data_in} + T_{comp} + T_{data_out} \cdots if(T_{data_out} < T_{data_in}) \quad (5)$$

$$T_{total_ioi_ps2} = T_{data_in} + T_{comp} + N_{process} T_{data_out} \cdots if(T_{data_out} \geq T_{data_in}) \quad (6)$$

To combine Equation (5) and (6) together, we derive Equation (7) as the following.

$$T_{total_ioi_ps2} = N_{process} \text{Max}(T_{data_in}, T_{data_out}) + T_{comp} + \text{Min}(T_{data_in}, T_{data_out}) \quad (7)$$

Even with the nature of no possible T_{comp} overlapping under PS-2, it still can achieve better performance by I/O overlapping for IO-I kernels compared with PS-1. By also comparing Equation (4) with (7), we are able to demonstrate that PS-2 brings better performance improvement specifically for the case of IO-I kernels. Therefore, our further analyses and implementation adopt PS-2 for IO-I kernels and PS-1 for C-I kernels specifically.

To estimate the potential theoretical performance gain of both C-I and IO-I kernel cases, we compare Equation (2) and (7) with Equation (1) and therefore derive the speedups as the following two equations.

$$S_{ci} = \frac{T_{total_no_vt}}{T_{total_ci_ps1}} = \frac{\left(N_{process}(T_{init} + T_{data_in} + T_{comp} + T_{data_out}) + (N_{process} - 1) T_{ctx_switch} \right)}{N_{process}(T_{data_in} + T_{data_out}) + T_{comp}} \quad (8)$$

$$S_{ioi} = \frac{T_{total_no_vt}}{T_{total_ioi_ps2}} = \frac{\left(N_{process}(T_{init} + T_{data_in} + T_{comp} + T_{data_out}) + (N_{process} - 1) T_{ctx_switch} \right)}{\left(N_{process} \text{Max}(T_{data_in}, T_{data_out}) + T_{comp} + \text{Min}(T_{data_in}, T_{data_out}) \right)} \quad (9)$$

As we mentioned earlier, our main purpose with the execution model is to estimate the optimal performance upper bound with GPU virtualization. With Equation (8) and (9), the theoretical maximum speedup with GPU virtualization can be derived by considering $N_{process}$ to infinity. If we take the limit of (8) and (9), the speedup upper bounds for both kernel cases can be derived in Equation (10) and (11).

$$S_{max_ci} = \lim_{N_{process} \rightarrow +\infty} S_{ci} = \frac{\left(T_{init} + T_{data_in} + T_{comp} + T_{data_out} \right)}{T_{data_in} + T_{data_out}} \quad (10)$$

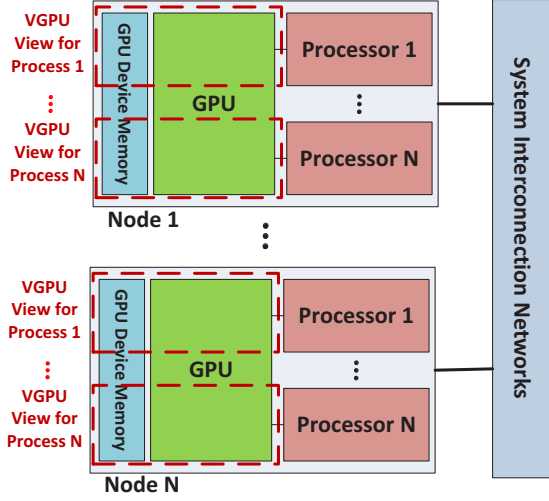


Figure 11: A Virtual SPMD View of Asymmetric Heterogeneous Cluster with GPU Virtualization

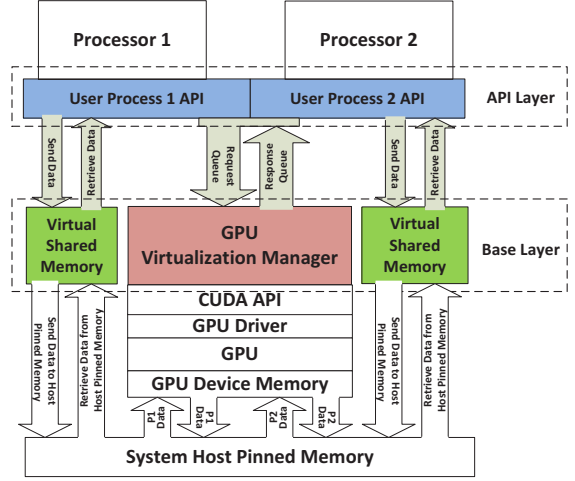


Figure 12: A Hierarchical View of the GPU Virtualization Infrastructure with Data Flow and Synchronization

$$S_{max_ioi} = \lim_{N_{process} \rightarrow +\infty} S_{ioi} = \frac{(T_{init} + T_{data_in} + T_{comp} + T_{data_out} + T_{ctx_switch})}{Max(T_{data_in}, T_{data_out})} \quad (11)$$

Equation (10) shows that the best theoretical speedup is achieved through limiting total T_{comp} through concurrent kernel execution under PS-1 as well as eliminating both initialization and context-switch overhead for C-I kernels. Equation (11) also demonstrates that I/O-I kernels can achieve the optimal performance gain with concurrent I/O under PS-2 as well as the elimination of both T_{init} and T_{ctx_switch} overheads with our virtualization technique, the implementation details of which is to be discussed in the following.

5. The GPU Virtualization Infrastructure

As we theoretically modeled and analyzed process-level SPMD execution parallelism and overlapping behaviors under the sharing scheme provided by GPU virtualization. In this section, we layout the GPU virtualization implementation. Figure 11 shows a representative HPC system architecture with several nodes connected by an interconnection network. Within each node there is a heterogeneous asymmetry between the microprocessors and GPU. Under the SPMD scenario, each processor runs the same application on different data while the application has a few computationally intensive functions (GPU kernels) to be executed on the GPU. The proposed GPU virtualization infrastructure is implemented as a virtualization layer that provides a virtual view of the GPU to each processor. In other words, each of the processes running on processors sees its own Virtual GPU (VGPU) and execute its own GPU kernel and data. The deployment of the virtualization infrastructure on each computing node establishes the virtual 1 to 1 CPU/VGPU ratio and eliminates potential resource underutilization with efficient GPU sharing.

Figure 12 shows the hierarchical view of the virtualization infrastructure and data flows between layers. The base layer of the infrastructure manages the underlying GPU computing and memory resources. On top of the base layer lies the user process API layer. The base layer is consisted of the GPU Virtualization Manager (GVM), virtual shared memory space for each processor as well as request/response message queues. The API layer, which provides a virtual GPU abstraction to each of the user processes, physically handles the inter-layer communication as well as data-transfer.

In the base layer, the GVM is a run-time process responsible for initializing all virtualization resources, handling requests from processes and processing requests on the GPU device. The initialization of the GVM

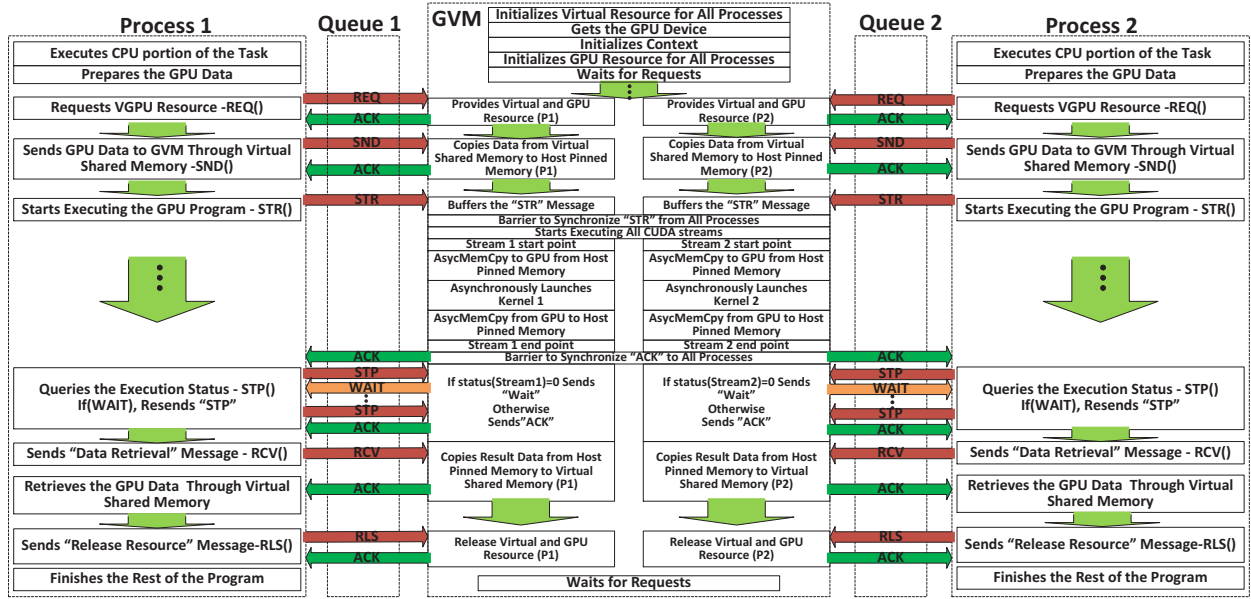


Figure 13: The Detailed Execution and Synchronization Flow of the GVM and Two User Processes

creates the virtualization resources including the virtual shared memory spaces and the request/response queues. The virtual shared memory space is implemented as POSIX shared memories for each of the processes so that each process has its own virtual memory space. The virtual shared memory space handles data exchanges between processes and the GVM. Furthermore, the shared memory size is user-customizable to ensure the total size does not exceed the GPU memory size. The request/response queues are implemented as POSIX message queues to stream the process requests into the GVM and provide handshaking synchronization responses. By using streaming queues, resource contention problems are prevented. The GVM also sets request barriers to ensure that SPMD tasks from different processes can be executed in parallel. On the GPU side, when initialized, the GVM creates the necessary GPU resources including the only required GPU context and CUDA streams for each process. It also creates different memory objects for each process separately to ensure data from different processes can co-exist in the GPU memory safely. These memory objects include both GPU device memory and host pinned memory. While host pinned memory provides better I/O bandwidth, it is also required to be setup for achieving concurrent I/O and kernel execution using asynchronous streams. Moreover, the GVM takes the requested CUDA kernel functions and prepares the kernels to be executed when initialized.

The abstraction of the API layer allows the user processes to interact with the underlying run-time virtualization layer in a transparent way. While the API layer exposes the user processes (the programmers) with a virtual GPU resource view, the programmers will only need to provide the base layer with the GPU kernel function they wish to execute on their individual VGPUs. The programmer also need to take care of data exchange with the virtual shared memory, by following the procedure shown in Figure 13 using our provided API routines such as `SND()`-send data routine, `STR()`-start execution routine and `RCV()`-receive data routine etc. Thus it requires very little effort to port existing GPU programs into the virtualization infrastructure.

When an SPMD program executes under GPU virtualization, the GVM initially sets up all virtual resources associated with each process and gets ready to serve process requests. All processes accordingly send requests of VGPU resource using `REQ()` function while all requests are queued into the GVM to be processed. The GVM always sends the ACK signal notifying the process if the requests have been served. Each user process then continues sending data to its own share memory space and using `SND()` function to request GVM to process the input data from the shared memory space. Once acknowledged, the process

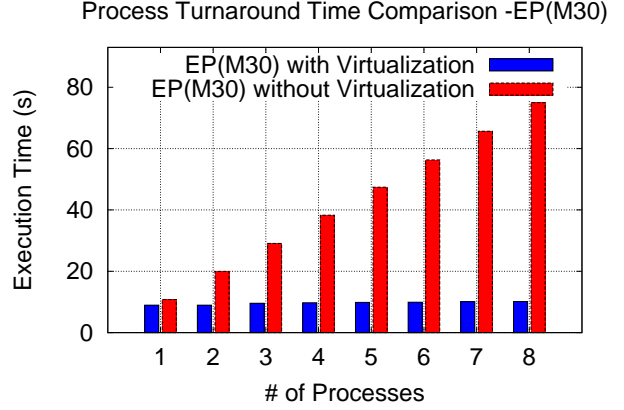
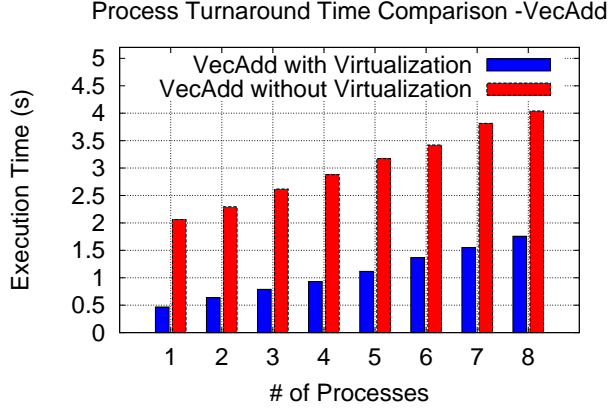


Figure 14: Process Turnaround Time Comparison: C-I
Figure 15: Process Turnaround Time Comparison: IO-I

continues to send kernel execution request through STR() followed by the query API - STP() until the ACK signal comes in (indicating that the result data is ready to be picked up from the shared memory space). Then the user process copies the result data back from the shared memory space through RCV() routine followed by using RLS() routine to request GVM to release all the VGPU resources associated with the process itself. The detailed execution flows and interaction of the two layer are demonstrated in Figure 13. Here we use a two-process SPMD program example to demonstrate the interaction and synchronization between the processes and the GVM. Since requests from multiple processes come roughly simultaneously under SPMD scenario, we also set necessary execution barriers in the GVM to flush multiple CUDA streams simultaneously. We also note that different stream programming styles are used according to our previous analytical results. In other words, inside the GVM, Compute-Intensive kernels are executed with PS-1 while PS-2 is adopted by I/O-Intensive kernels for the possibly optimal concurrencies and overlapping.

6. Experiments and Performance Evaluation

In this section, in order to demonstrate that using the proposed GPU virtualization approach and infrastructure can provide effective resource sharing and performance gains under the SPMD execution, we conduct a series of experiments. Here we use different emulated process-level SPMD benchmarks, which launch the same benchmark program in different processes, with the affinity of each process set to a unique CPU core. We essentially conduct the experiments to compare with the same emulated process-level SPMD programs without virtualization. In other words, the performance when each process shares the GPU natively, as the performance baseline we modeled. We are mostly interested in comparing the process turnaround time, which is the time for all processes to finish executing the benchmarks after they start simultaneously.

The experiments are conducted on our GPU computing node, which is equipped with dual Intel Xeon X5570 quad-core processors (eight cores in total) running at 2.93 GHz, 48GB system memory and the NVIDIA Tesla C2070 GPU with 6GB device memory. Tesla C2070 consists of 14 SMs running at 1.15GHz and allows maximum 16 concurrent running kernels. Both CUDA driver and SDK versions are 5.0, which run under Ubuntu 12.04 with 3.2.0-23 Linux kernel.

Our previous execution models depict inter-process parallelism and overlapping under the sharing scheme achieved through GPU virtualization. As we use two kernel cases to analyze the overlapping behaviors in the execution model, here we first utilize two extreme benchmark cases (Highly Compute-Intensive and Highly I/O-Intensive) as experimental evaluation of potential overlapping behavior. The purpose is to demonstrate the different overlapping behavior for C-I and IO-I kernel cases and compare the actual performance gain with

Table 3: GPU Virtualization Benchmark Profiles

Benchmark	Problem Size	Grid Size	Class
NPB EP(M30)	M=30	4	Compute-Intensive
Vector Addition (VecAdd)	50M Float	50K	I/O-Intensive
NPB EP(M24)	M=24	1	Compute-Intensive
Vector Multiplication (VecMul)	16M Float / 15 Iters	16K	I/O-Intensive
Matrix Multiplication (MM)	2Kx2K Matrix	4K	Intermediate
NPB MG	S (32x32x32 / 4 Iters)	64	Compute-Intensive
BlackScholes (BS)	1M Calls / 512 Iters	480	I/O-Intensive
NPB CG	S (NA=1400 / 15 Iters)	8	Compute-Intensive
Electrostatics (ES)	100K Atoms / 25 Iters	288	Compute-Intensive

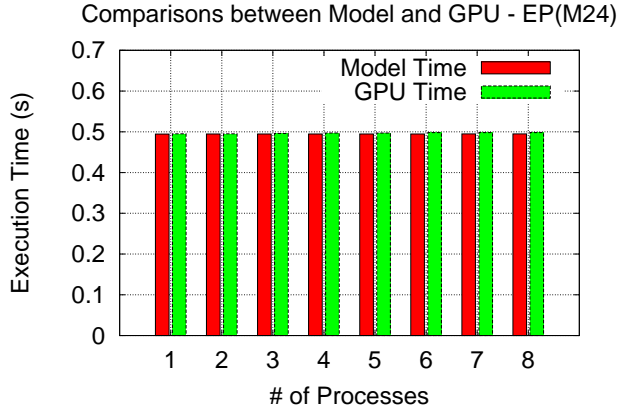


Figure 16: Execution Model Validation: C-I

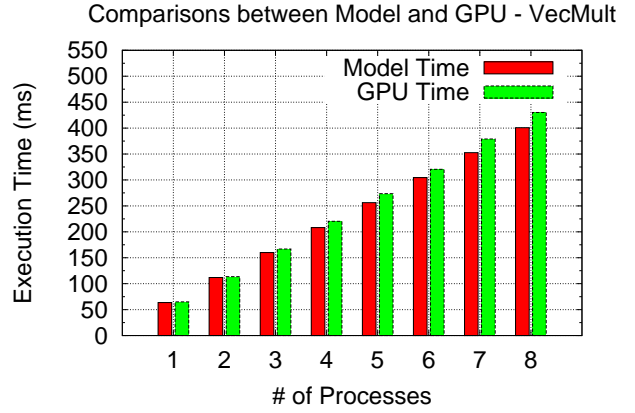


Figure 17: Execution Model Validation: IO-I

non-virtualization solution. The I/O-Intensive application we use is a very large vector addition benchmark while the Compute-Intensive benchmark is the GPU version [38] of EP (Problem Size: M=30) from NAS Parallel Benchmarks (NPB) [39]. The EP kernel grid size is designed small merely to show the effectiveness of concurrency under virtualization, while the actual grid size decides the overlapping and concurrency extent in real applications. We list all benchmark kernel profiles used in this section in Table 3.

Our experiment with EP(M30) and VecAdd primarily focuses on evaluating process turnaround time by emulating a process-level parallel SPMD program for both benchmarks, while launching multiple processes with the same benchmark kernel simultaneously. As an SPMD program generally requires $N_{process} \leq N_{processor}$ and our computing node consists of 8 microprocessor cores, we varied the number of SPMD parallel processes from 1 to the maximum of 8. Figure 14 and 15 demonstrate the effectiveness of GPU virtualization in reducing the process turnaround time with the increasing number of processes for both C-I and IO-I cases. For the I/O-Intensive benchmark in Figure 14, when the number of parallel processes increases, without virtualization, the turnaround time increase sharply due to the zero overlapping and context-switch overheads. With virtualization, the turnaround time still increases but much slowly comparatively. This is because I/O-Intensive application cannot achieve complete overlapping as explained in the model earlier, but can still partially overlap I/O as well as eliminate context-switch and initialization overheads. For the Compute-Intensive benchmark in Figure 15, with virtualization, the turnaround time increases very little with the increasing number of processes, which clearly shows that our GPU virtualization implementation can achieve the expected execution concurrencies for smaller kernels only using a portion of the GPU resource in the case of Compute-Intensive kernel.

CUDA's current concurrent kernel execution support heavily depends on kernel profiles. In other words,

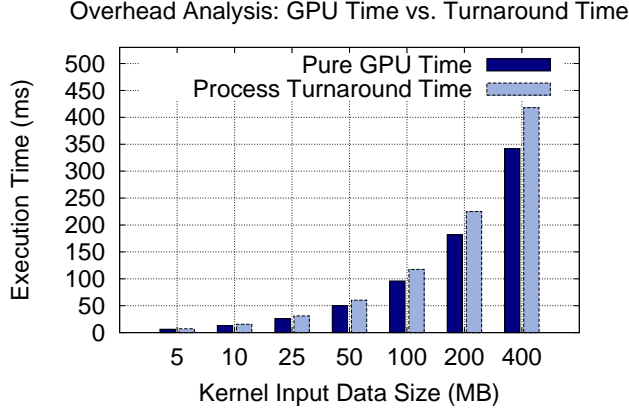


Figure 18: Virtualization Overhead

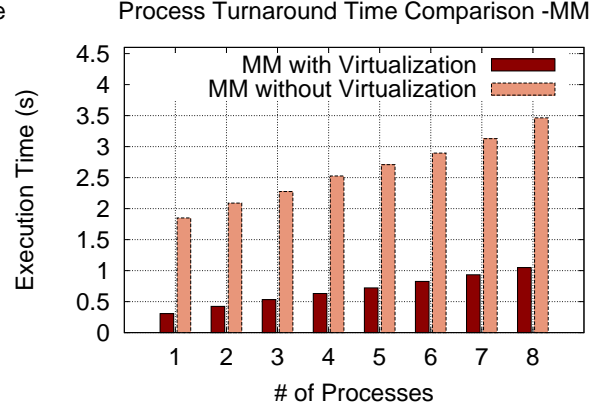


Figure 19: Performance: MM

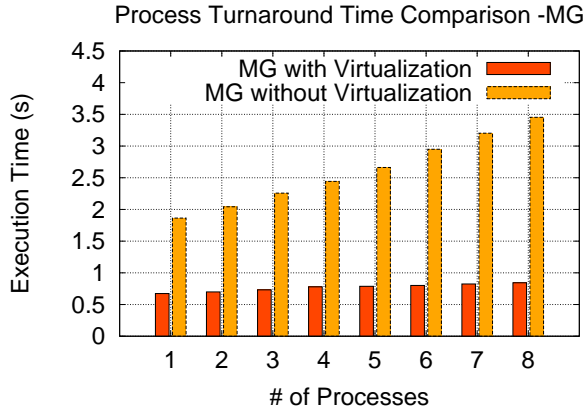


Figure 20: Performance: MG

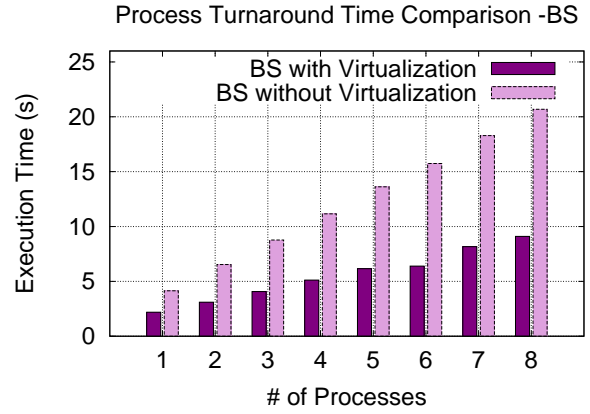


Figure 21: Performance: BS

blocks from multiple kernels are concurrently executed on separated SMs inside GPU to achieve the concurrency when CUDA streams are used. Thus small kernels (small number of blocks) can achieve better kernel execution concurrency compared with large kernels. In the previous modeling analysis, we assume kernel execution overlapping is complete since we are focused on studying overlapping behaviors. Thus in order to verify our previous modeling analysis, we utilize EP(M24) and VecMult shown in Table 3 as the benchmark kernels to verify C-I and IO-I models, respectively. For both kernels, we carry out initial profiling analysis to empirically derive T_{data_in} , T_{comp} and T_{data_out} . As both execution models are to estimate the total execution time of $N_{process}$ kernels sharing the GPU under virtualization. The theoretical time can be derived using Equation (2) and (7) respectively with the profiling results. Experimentally, we here launch the emulated SPMD kernel programs while varying $N_{process}$ from 1 to 8. Instead of measuring process turnaround time, we here only measure the time all kernels spend on sharing the GPU inside the GVM of our virtualization infrastructure. Thus, we are able to avoid bringing unnecessary virtualization overheads into the model validation. Comparisons of experimental results and modeling results are shown in Figure 16 for C-I model and in Figure 17 for IO-I model, both of which demonstrate accurate modeling results. We also note that for C-I model validation, utilizing EP(M24) with kernel size of one is merely to guarantee that all kernels are executed on separated SMs (up to 8 kernels in our case). In other words, complete overlapping of actual kernel computation can be achieved with kernels executed on separated SMs. The comparisons in both figures validate our previous execution model results with an average model deviation

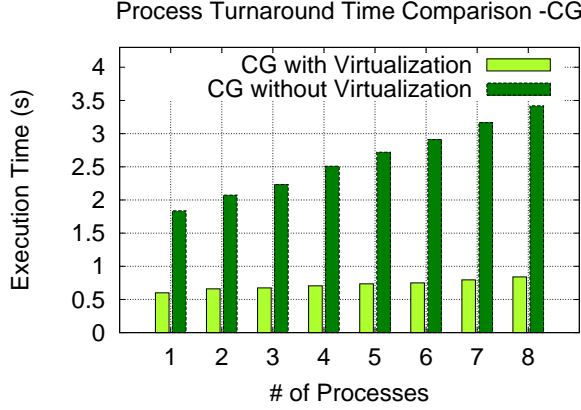


Figure 22: Performance: CG

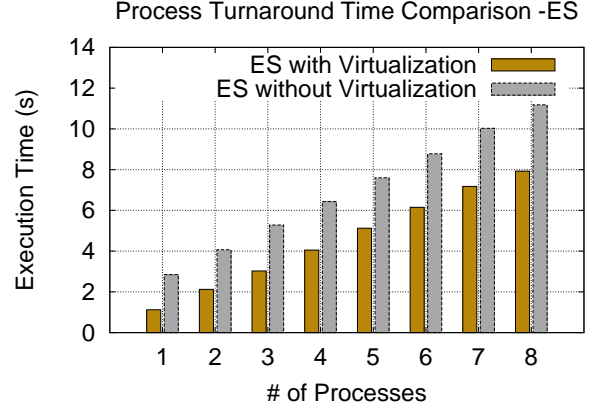


Figure 23: Performance: ES

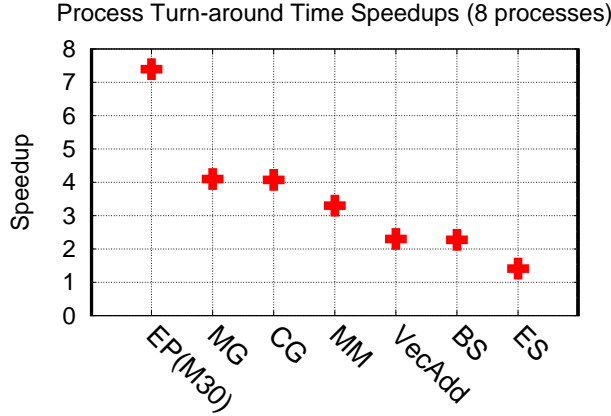


Figure 24: Virtualization Speedups

of 0.42% for EP(M24) and 4.76% for VecMult.

Considering the virtualization infrastructure is an add-on layer, possible overhead can be added on top of the theoretical modeling results. As our implementation mainly uses POSIX shared memory and message queue, the vast majority overhead comes from data movement and message synchronization between the API layer and base layer. We here conduct another micro benchmark using the I/O-Intensive VecAdd benchmark with multiple data sizes. We measure the overhead by launching a single process and compare the time purely spent on the GPU in the base layer with the process turnaround time. As shown in Figure 18, the overhead, which is the differences between the turnaround time and pure GPU time, increases with the size of data being transferred as expected. Even in the case when the data size is very large (400MB in our case), the virtualization overhead is measured around 20%, which demonstrates that our virtualization implementation incurs comparatively low overhead, especially considering that an add-on virtualization layer generally brings much more overhead.

As a further step, we conduct several additional benchmarks to demonstrate the efficiency of the proposed virtualization approach in addressing real-life applications with different profiles. As Table 3 shows, MM refers to the 2048x2048 single precision floating-point matrix multiplication. MG and CG refer to GPU versions [38] of NPB [39] kernel MG and CG, respectively, with the problem size of Class S. Black Scholes [40] is a European option pricing benchmark used in financial area, adapted from NVIDIA’s CUDA SDK. We set option prices over 512 iterations as default. Electrostatics refers to fast molecular electrostatics

algorithm as a part of the molecular visualization program VMD [41] and we set the problem size to be 100K atoms with 25 iterations. By evaluating I/O and computing time ratio, we further profile the class of each benchmark as shown in Table 3. Experimentally, we emulate process-level SPMD execution of each benchmark kernel with multiple processes and compare the process turnaround time between virtualization and non-virtualization scenario. Figure 19, 20, 21, 22 and 23 respectively compare the turnaround time with and without GPU virtualization. It is worth mentioning that the performance improvement using one process is due to the elimination of initialization overheads by the virtualization implementation, even with the add-on virtualization overhead. Since MM is profiled as intermediate and the grid size is large enough to occupy the whole GPU, it partially benefits from both I/O and kernel computing overlapping with virtualization. Both MG and CG are Compute-Intensive benchmarks and Class S problem sizes (small kernel sizes) only make MG and CG utilize partial GPU resource. Thus MG and CG can achieve more overlapping by concurrent kernel execution under virtualization. With the default problem size and a grid size of 480, a single Black Scholes benchmark can utilize full GPU resource and can hardly be concurrently executed under virtualization. Since it is also I/O-Intensive application, it is only able to achieve limited overlapping between the I/O and kernel-computing as described earlier. As for Electrostatic benchmark, since it is Compute-Intensive while the grid size of 288 making it occupy the whole GPU, the overlapping potential is small using virtualization. However, it still benefits from zero context-switch and initialization overhead due to virtualization. Therefore, within the five application benchmarks, as each achieves certain amount of performance gain through virtualization due to overlapping and elimination of overheads, MG and CG achieve better performance gains.

Figure 24 summarizes an example speedup comparison scenario utilizing all available system processors (8 processes). Including EP(M30) and VecAdd as two extreme cases along with the five real-life benchmarks, all seven benchmarks we conduct achieve speedups from 1.4 to 7.4 with 8 process-level parallelism under GPU virtualization. Therefore, while all benchmarks can achieve certain amount of performance gains, the efficiency of the virtualization approach also depends on the profiles of the applications, including the I/O and computing time ratio as well the GPU resource usage. To summarize from Figure 24, small Compute-Intensive kernels can achieve the best performance improvement as EP(M30), MG and CG show. Intermediate kernels can achieve reasonable speedups with partial I/O and compute overlapping as shown from MM. I/O-Intensive kernels (BS and VecAdd) can only achieve I/O overlapping, while large Compute-Intensive kernels (ES) can overlap I/O (small portion) and very limited kernel execution. Thus they achieve relatively less performance gain. However, the elimination of context-switch and initialization overhead plus well-increased GPU utilization from GPU virtualization allow considerable speedup for application in general. In fact, our virtualization experimental results show a good agreement with the proposed analytical model, and demonstrate that our GPU virtualization implementation is an effective approach allowing multi-processes to share the GPU resource efficiently under SPMD model, while incurring comparatively low overhead.

7. Conclusion

In this paper, we proposed a GPU virtualization approach which enables efficient sharing of GPU resources among microprocessors in heterogeneous HPC systems under SPMD execution model. In achieving the desired objective of making each microprocessors effectively utilize shared resources, we investigated the concurrency and overlapping potentials that can be exploited on the GPU device-level. We also analyzed the performance and overheads of direct GPU access and sharing from multiple microprocessors as a comparison baseline. We further provided an analytical execution model as a theoretical performance estimate of our proposed virtualization approach. The analytical model also provided us with better understanding of the methodologies in implementing our virtualization concept. Based on these concepts and analyses, we implemented our virtualization infrastructure as a run-time layer running in the user space of the OS. The virtualization layer manages requests from all microprocessors and provides necessary GPU resources to the microprocessors. It also exposes a VGPU view to all the microprocessors as if each microprocessor has its own GPU resource. Inside the virtualization layer, we managed to eliminate unnecessary overheads and achieve possible overlapping and concurrency of executions. In the experiments, we utilized our NVIDIA Fermi

GPU computing node as the test bed. We used initial I/O-Intensive and Compute-Intensive benchmarks as well as application benchmarks with multiple folds of analyses in our experiments. Our experimental results showed that we were able to achieve considerable performance gains in terms of speedups with our virtualization infrastructure with low overhead. The experimental results also demonstrate an agreement with our theoretical analysis. Proposed as a solution for microprocessor resource underutilization by providing a virtual SPMD execution scenario, our approach proves to be effective and efficient and can be deployed to any heterogeneous GPU clusters with imbalanced CPU/GPU resources.

References

- [1] T. Li, V. K. Narayana, E. El-Araby, T. El-Ghazawi, GPU resource sharing and virtualization on high performance computing systems, in: *Parallel Processing (ICPP)*, 2011 International Conference on, IEEE, 2011, pp. 733–742. doi:10.1109/ICPP.2011.88.
- [2] GPGPU webpage, 2013.
- [3] NVIDIA Corp., *Tesla GPU Computing Brochure*, 2010.
- [4] AMD, *AMD FirePro S10000 Datasheet*, 2013.
- [5] Z. Fan, F. Qiu, A. Kaufman, S. Yoakum-Stover, GPU cluster for high performance computing, in: *Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, IEEE Computer Society, 2004, p. 47.
- [6] V. Kindratenko, J. Enos, G. Shi, M. Showerman, G. Arnold, J. Stone, J. Phillips, W. Hwu, GPU clusters for high-performance computing, in: *Cluster Computing and Workshops, 2009. CLUSTER'09. IEEE International Conference on*, IEEE, 2009, pp. 1–8.
- [7] Cray Inc., *Cray XK7 Brochure*, 2012.
- [8] SGI Corp., *SGI GPU Compute Solutions*, 2012.
- [9] Titan Webpage in Oak Ridge National Lab, 2013.
- [10] Top 500 Supercomputer Sites Webpage, 2013.
- [11] F. Damera, The SPMD model: Past, present and future, *Recent Advances in Parallel Virtual Machine and Message Passing Interface (2001)* 1–1.
- [12] W. Gropp, E. Lusk, A. Skjellum, *Using MPI: portable parallel programming with the message passing interface*, volume 1, MIT press, 1999.
- [13] China National Supercomputer Center in Tianjin Webpage, 2013.
- [14] Nebulae Specification in Sugon Webpage, 2013.
- [15] GSIC, Tokyo Institute of Technology., *Tsubame Hardware Software Specifications*, 2011.
- [16] Advanced Micro Devices, Inc., *AMD Firestream 9350 Datasheet*, 2011.
- [17] NVIDIA Corp., *NVIDIA's Next Generation CUDA Compute Architecture: Fermi*, 2009. Ver. 1.1.
- [18] NVIDIA Corp., *NVIDIA CUDA C-Programming Guide V5.5*, 2013.
- [19] V. Gupta, A. Gavrilovska, K. Schwan, H. Kharche, N. Tolia, V. Talwar, P. Ranganathan, GViM: GPU-accelerated Virtual Machines, in: *Proceedings of the 3rd ACM Workshop on System-level Virtualization for High Performance Computing*, ACM, 2009, pp. 17–24.
- [20] L. Shi, H. Chen, J. Sun, vCUDA: GPU accelerated high performance computing in virtual machines, in: *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, IEEE, 2009, pp. 1–11.
- [21] G. Giunta, R. Montella, G. Agrillo, G. Coviello, A GPGPU transparent virtualization component for high performance computing clouds, *Euro-Par 2010-Parallel Processing (2010)* 379–391.
- [22] J. Duato, F. Igual, R. Mayo, A. Peña, E. Quintana-Ortí, F. Silla, An efficient implementation of GPU virtualization in high performance clusters, in: *Euro-Par 2009-Parallel Processing Workshops*, Springer, 2010, pp. 385–394.
- [23] M. Guevara, C. Gregg, K. Hazelwood, K. Skadron, Enabling task parallelism in the CUDA scheduler, in: *Proceedings of the Workshop on Programming Models for Emerging Architectures (PMEA)*, Citeseer, 2009.
- [24] A. Saba, R. Mangharam, Anytime Algorithms for GPU Architectures, *AVICPS 2010 (2010)* 31.
- [25] T. Li, V. K. Narayana, T. El-Ghazawi, A static task scheduling framework for independent tasks accelerated using a shared graphics processing unit, in: *Parallel and Distributed Systems (ICPADS)*, 2011 IEEE 17th International Conference on, IEEE, 2011, pp. 88–95. doi:10.1109/ICPADS.2011.13.
- [26] T. Li, V. K. Narayana, T. El-Ghazawi, Symbiotic scheduling of concurrent GPU kernels for performance and energy optimizations, in: *Proceedings of the 11th ACM Conference on Computing Frontiers, CF '14*, ACM, New York, NY, USA, 2014, pp. 36:1–36:10. URL: <http://doi.acm.org/10.1145/2597917.2597925>. doi:10.1145/2597917.2597925.
- [27] T. Li, V. K. Narayana, T. El-Ghazawi, A power-aware symbiotic scheduling algorithm for concurrent gpu kernels, in: *The 21st IEEE International Conference on Parallel and Distributed Systems (ICPADS 2015)*, IEEE, 2015.
- [28] V. Ravi, M. Becchi, G. Agrawal, S. Chakradhar, Supporting GPU sharing in cloud environments with a transparent runtime consolidation framework, in: *the International Symposium on High-Performance Parallel and Distributed Computing*, 2011.
- [29] H. Peters, M. Koper, N. Luttenberger, Efficiently using a CUDA-enabled GPU as shared resource, in: *Computer and Information Technology (CIT)*, 2010 IEEE 10th International Conference on, IEEE, 2010, pp. 1122–1127.
- [30] S.GPU project webpage, 2013.

- [31] T. Li, V. K. Narayana, T. El-Ghazawi, Accelerated high-performance computing through efficient multi-process GPU resource sharing, in: Proceedings of the 9th Conference on Computing Frontiers, CF '12, ACM, New York, NY, USA, 2012, pp. 269–272. URL: <http://doi.acm.org/10.1145/2212908.2212950>. doi:10.1145/2212908.2212950.
- [32] T. Li, V. K. Narayana, T. El-Ghazawi, Exploring graphics processing unit (GPU) resource sharing efficiency for high performance computing, Computers 2 (2013) 176–214. URL: <http://www.mdpi.com/2073-431X/2/4/176>. doi:10.3390/computers2040176.
- [33] T. Li, Efficient Virtualization and Scheduling for Productive GPU-based High Performance Computing Systems, Ph.D. thesis, The George Washington University, 2015.
- [34] C. Huang, P. Hsiung, Hardware resource virtualization for dynamically partially reconfigurable systems, Embedded Systems Letters, IEEE 1 (2009) 19–23.
- [35] T. Li, M. Huang, T. El-Ghazawi, H. Huang, Reconfigurable active drive: An fpga accelerated storage architecture for data-intensive applications, 2009 Symposium on Application Accelerators in High-Performance Computing (2009) 1–3.
- [36] E. El-Araby, I. Gonzalez, T. El-Ghazawi, Virtualizing and sharing reconfigurable resources in High-Performance Reconfigurable Computing systems, in: High-Performance Reconfigurable Computing Technology and Applications, 2008. HPRCTA 2008. Second International Workshop on, IEEE, 2008, pp. 1–8.
- [37] Khronos OpenCL Working Group, The OpenCL Specification V2.0, 2013.
- [38] M. Malik, T. Li, U. Sharif, R. Shahid, T. El-Ghazawi, G. Newby, Productivity of GPUs under different programming paradigms, Concurrency and Computation: Practice and Experience (2011) 179–191. URL: <http://dx.doi.org/10.1002/cpe.1860>. doi:10.1002/cpe.1860.
- [39] D. Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, P. Frederickson, T. Lasinski, R. Schreiber, et al., The NAS parallel benchmarks, International Journal of High Performance Computing Applications 5 (1991) 63.
- [40] F. Black, M. Scholes, The pricing of options and corporate liabilities, The journal of political economy (1973) 637–654.
- [41] Visual molecular dynamics program webpage, 2013.